Privacy-Preserving and Trustworthy Cyber-Infrastructures

CIS 4212/6214 (Spring 2026)

Instructor: **Dr. Attila A. Yavuz**
TA: ***Saif E. Nouma***

Homework 2 (Due: Feb 15^{th}, 2026, 23:59)[100 Points + 30 Extra Points]

(Upload your solutions to the CANVAS)
Requirements: HW groups are 2-3 people. Each student should contribute all parts of the answers. If one group member drops, the others are equally responsible (each HW can be handled by a single student if needed, so no excuse for missing a partner (if you are doing in group of 2), see syllabus for breaking-up with pair programming partner). Directly borrowing (e.g., copy-paste) from any material and putting in solutions (e.g, from online solutions, AI tools, Wikipedia, or research papers) is plagiarism (see Syllabus for its corresponding actions). Please cite very carefully each resource you use but citing a solution does not give a license to directly put it as an answer. **All of your answers must be in your own words and interpretations.** Using AI tools like ChatGPT, GitHub Copilot, or similar to generate code constitutes cheating. **All your solutions will be passed through MOSS and Turnitin for plagiarism check.**
Note: Many of these questions require you to recall explanations/discussions during the class (this is why attendance is important), and to also research on the internet to find good answers. You may also use AI (e.g., the AI Companion tool, ChatGPT) to enhance your experiences (e.g., asking for more information or a summary of the materials). Please note that very brief and/or informal explanations will potentially be not sufficient.
Remark: HW should be prepared by a text editor (e.g., Microsoft Word or Latex). Handwritten submissions are not accepted. In your answer key, please indicate the point of each question as is done in this HW document. Preferably, take questions from here, plug them into your answer key with their sub scores, and provide your answer below them. This will facilitate grading.
As indicated in Syllabus, **no partial credit is possible for coding answers that does not produce correct results with respect to test vectors, deviate from the description and rubric, are coded on platforms other than specified, or do not work on the grader's machine.**

0.1 Programing Assignment - Client-Server Puzzles for DoS Prevention. [50 points]

This assignment will require you to implement a **Client-Server Puzzle Protocol** to counter Denial of Service (DoS) attacks, based on the **broadcast puzzle construction** from lecture (slide 21). Recall previous lectures on Counter DoS techniques, client puzzles, and hash-based cryptographic primitives. This assignment will put those concepts into practice, where you will implement puzzle generation, puzzle solving, and solution verification using SHA256 from OpenSSL.

Implementation Setting

- Puzzle Type: Proof-of-work (find k leading zero bits).
- To ensure the accuracy of your code, utilize unsigned char for raw data and hex strings for file I/O. So, perform the conversions accordingly.

- You have access to all the functions provided in class exercises or homework assignments. Please modify them as necessary. Additionally, for your convenience, we have provided a template that contains all the necessary functions and their description so that you could use them to create your solution (**"UsefulFunctions.c"**).
- Also, feel free to use any function of your choice as long as it produces the correct output. Refer to the test vectors to understand the grading criteria.

Protocol Overview

k **Leading Zeros Puzzle Construction.** Please refer to Figure 1 for further depiction.

Server → All clients:
- Challenge (timestamp || server_nonce)
- Difficulty k

Client → Server:
- Nonce such that SHA256(challenge || nonce) has k leading zeros

Server verifies:
- Recompute SHA256(challenge || nonce)
- Check if hash has k leading zero bits
- Accept if valid, Reject otherwise

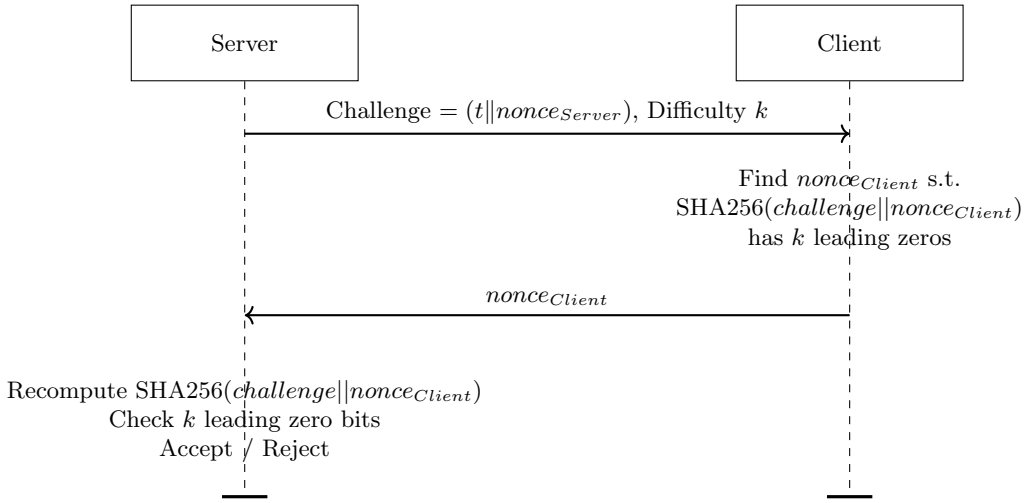


Figure 1: Client–Server Proof-of-Work Challenge-Response Protocol

Expected Solving Time: Client tries nonces: 0, 1, 2, 3, $2^k - 1$ ... Average iterations: 2^{k-1}

Example: $k = 12 \rightarrow$ 2048 iterations on average

Puzzle Generation: Server [10 points]

The server generates and broadcasts a puzzle challenge to clients:

1. Server reads the challenge data from a file named **"Challenge$i.txt"** (32 bytes hex).

- Technically, this contains timestamp || server_nonce
- Read the challenge as char so that you can use the provided functions

2. Server reads the difficulty level from a file named **"Difficulty$i.txt"** (integer).
   - Value between 8 and 20 representing k-bit difficulty
   - Read as ASCII integer
3. Server writes the challenge to **"puzzle_challenge.txt"** as hex string.
4. Server writes the difficulty k to **"puzzle_k.txt"** as ASCII integer.

Terminal
Compile the puzzle generation part with the following command:
gcc server.c -lcrypto -o Server
Run your code for the first test files with the following commands:
./Server Challenge1.txt Difficulty1.txt

Puzzle Solving: Client [20 points]

Client Algorithm:

Input: Challenge, difficulty k
Output: Nonce with k leading zeros

```
1) Read challenge from puzzle_challenge.txt
2) Read difficulty k from puzzle_k.txt
3) For nonce = 0 to 2^k − 1:
       data = challenge || nonce
       hash = SHA256(data)
       if hash has k leading zero bits:
           return nonce
4) Write nonce to solution_nonce.txt
5) Write iteration count to solution_iterations.txt
```

Client performs puzzle solving as follows:

1. Client reads the puzzle challenge from file **"puzzle_challenge.txt"**.
   - Read the challenge as char, so that you can easily convert it from hex to bytes.
2. Client reads the difficulty k from file **"puzzle_k.txt"**.
   - Read k as ASCII integer.
3. Client performs brute-force search:
   - Starting from nonce = 0, try incrementing values
   - For each nonce:
     - Construct data = challenge || nonce
     - Compute hash = SHA256(data)
     - Check if first k bits are zero

- Stop when solution found

4. To check for k leading zero bits:
   - Full zero bytes: $k/8$
   - Partial bits: $k \bmod 8$
   - Use bit masking to check partial bits
5. After finding the solution, client writes the nonce (8 bytes) to **"solution_nonce.txt"** as hex.
6. Client writes the iteration count to **"solution_iterations.txt"** as ASCII integer.

Terminal
Compile the puzzle solving part with the following command:
gcc client.c -lcrypto -o Client
Run your code with the following commands:
./Client puzzle_challenge.txt puzzle_k.txt
In total, Client should produce 2 files: **solution_nonce.txt** and **solution_iterations.txt**

Solution Verification: Server [20 points]

Verification Algorithm:

Input: Challenge, difficulty k, solution nonce
Output: Accept or Reject

1) Read challenge from puzzle_challenge.txt
2) Read difficulty k from puzzle_k.txt
3) Read nonce from solution_nonce.txt
4) Construct data = challenge || nonce
5) Compute hash = SHA256(data)
6) Check if hash has k leading zero bits
7) If yes: Write "ACCEPT", exit 0
8) If no: Write "REJECT", exit 1

Server performs solution verification as follows:

1. Server reads the puzzle challenge from file **"puzzle_challenge.txt"**.
   - Read the challenge as char, so that you can easily convert it from hex to bytes.
2. Server reads the difficulty k from file **"puzzle_k.txt"**.
   - Read k as ASCII integer.
3. Server reads the solution nonce from file **"solution_nonce.txt"**.
   - Read nonce as char, so that you can easily convert it from hex to bytes.
4. Server recomputes the hash:
   - Construct data = challenge || nonce
   - Compute hash = SHA256(data)
5. Server verifies leading zeros:

- Check if hash has k leading zero bits
- Use same checking logic as client

6. Server outputs result:
   - If valid: Write **"ACCEPT"** to **"verification_result.txt"**, exit 0
   - If invalid: Write **"REJECT"** to **"verification_result.txt"**, exit 1

Terminal
Compile the verification part with the following command:
gcc verify.c -lcrypto -o Verify
Run your code with the following commands:
./Verify puzzle_challenge.txt puzzle_k.txt solution_nonce.txt

Notes:

- An invalid solution must cause Verify to exit and write "REJECT". If Verify accepts an incorrect solution, you will lose credit.
- Use SHA256 from OpenSSL for all hashing operations.
- Leading zero check must be correct: the first k bits (from MSB) must be zero, not the last k bits.
- Nonce is 8 bytes (64-bit unsigned integer).
- We have provided a script (VerifyYourSolutionClientPuzzle.sh) and test vectors that you can use to test the correctness of your final code. Please, only submit your code if it is functional.
- In order to use the script, just put your codes, the rest of the provided files along with the "VerifyYourSolutionClientPuzzle.sh" in one folder and run the following command in the terminal:
  bash VerifyYourSolutionClientPuzzle.sh
- Ensure the output of your code is correctly named as per the instructions.
- You may look at the testing script to understand how your code is being tested. This script is the same one used for grading on our side, however the test vectors will be different.
- **You Must submit 3 files for this assignment:**
  server.c & client.c & verify.c

Rubric:

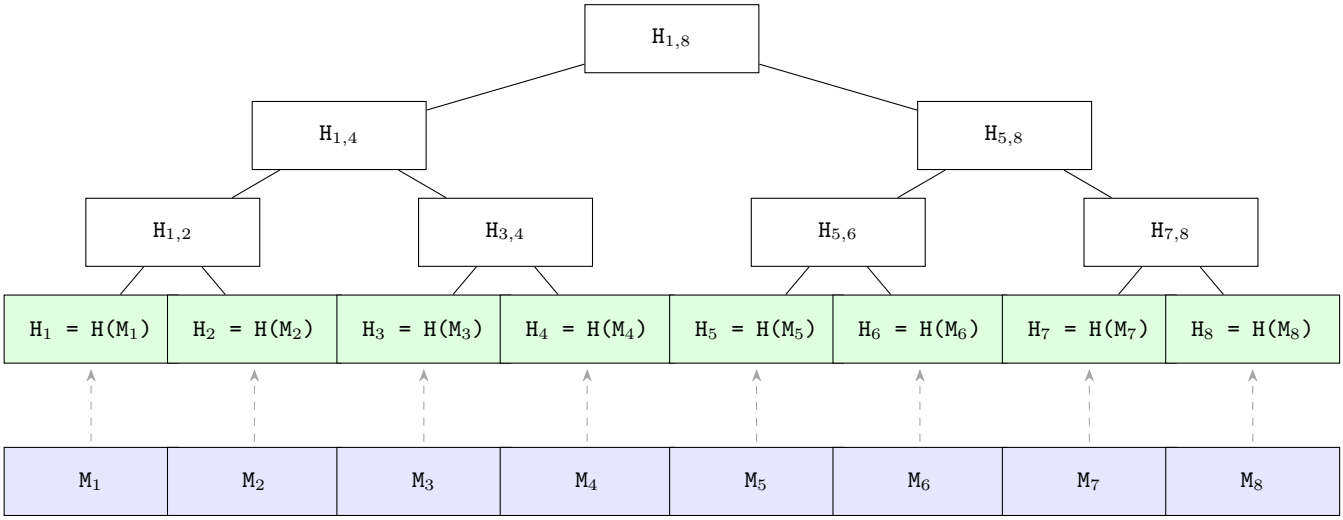
- The **"puzzle_challenge.txt"** file and the **"puzzle_k.txt"** files **[10 pts]**
- The **"solution_nonce.txt"** file and correct solving algorithm **[20 pt]**
- Verify **correctly accepting** valid solutions **[10 pt]**
- Verify **correctly rejecting** invalid solutions **[10 pt]**

0.2 Programming Assignment - Merkle Hash Tree (MHT slides 9-12) [50 pt]

This exercise will put basic cryptographic tools and Merkle Hash Tree (MHT) that were discussed during the class into action. Specifically, in this exercise, as shown in MHT slides, we will utilize again the SHA256 Hash function to create the root of MHT and perform the authentication by providing the path for each leaf. This exercise has two phases named "Offline" and "Online" as described below:

Offline Phase: [25 pt]

In this phase, you need to build the tree and compute the root for 8 messages. Please refer to the following figure for further depiction of the tree structure. The leaves of the Merkle hash tree (MHT) are the hash values of the eight messages. The tree is constructed iteratively by concatenating and hashing each pair of successive nodes at every level, proceeding until a single root hash is obtained.



1. First, you read the messages from a file named "Messages.txt". Specifically, this file has 8 lines of 256-bit random data (32 Bytes of random characters). (Read the message as unsigned char so that you could use the functions provided in in-class exercises.)
2. Then, you need to hash (SHA256) each message to create the leaves of the tree.
3. Afterward, concatenate the resulting hashes two-by-two to build the upper layer leaves.
4. Do the same concatenation and hashing so that you can get the root of the tree.
5. First, convert the root to a Hex string and then, write it in a file named "TheRoot.txt".

Online Phase: [25 pt]

In this phase, given an index of an element from the messages, you must be able to compute the path for the verification of that message.

1. Your program gets the index of the message as an input argument when running your code.
2. Based on the given index as an input argument, your program must be capable of computing the path for the verification of that index by MHT. Specifically, considering 8 elements as the bottom leaves of the tree, the path contains the required hash values for the verification process. For example, based on a tree with 8 leaves, by running 3 as an input index (i.e., M_3), your program must provide the values of H_4, $H_{1,2}$, and $H_{5,8}$ as the correct path.

3. Write the Hex values of the correct path to a file named "ThePath.txt". Technically, you need to find the path, then get their values, convert them to a Hex string, and then write them into the file in multiple lines.

As mentioned above, you need 2 files for the output:

1. The "**TheRoot.txt**" file contains the hex values of the computed root.
2. The "**ThePath.txt**" file contains the hex values of the correct path for the given index.

Notes:

- You require one file to run your code: "Messages.txt".
- For the correctness of reading from a file and XORing, please use unsigned char. The file contains 8 lines of 32 bytes random characters. You must read them line by line. You can use the provided functions in the template.
- We have provided a script and test files that you could use to test the correctness of your final codes (mht.c). Please, only submit your code if it is functional.
- We only test your code on a tree with 8 elements each of them 32 Bytes. However, in grading on our side, the messages will be chosen at random, and everything must work.
- You can check your answer with each provided test file manually or you can use the script to verify your solution for the whole HW2 programming question.
- In order to use the script, just put "mht.c" and the rest of the provide files along with the "VerifyYourSolutionHW2.sh" in one folder and run the following command in the terminal:
  bash VerifyYourSolutionMHT.sh
- Compile your codes with the following commands:
  gcc mht.c -o mht -lcrypto
- Run your codes for the first test files and M1 with the following commands:
  ./mht Messages1.txt M1
- **You Must submit one file for this assignment:**
  mht.c

Rubric:

- The **"TheRoot.txt"** file **[25 pt]**
- The **"ThePath.txt"** file **[25 pt]**

0.3 Extra Credit [30 points]

0.3.1 Cryptographic Puzzle Construction [10 points]

In the lecture slides, you saw that puzzles that either find k leading zeros, or partial pre-image with HMAC. You implemented the k leading zeros construction in this assignment.
Why is the k leading zeros construction simpler to implement and verify than a partial pre-image construction? What advantage does partial pre-image have over k leading zeros?

0.3.2 Client Server Puzzles (CounterDoS.ppt and its reference papers) [20 points]

A. Slide 8 describes four desirable properties of a basic counter DoS puzzle, and Slides 9-12 gave the main idea behind a puzzle construction that achieves these features (please see corresponding papers provided in the references if you need more details). With approximately 4-5 sentences each, explain how (X', Y)-puzzle design can achieve each of these desirable properties (remember in-class discussions explaining these in detail). Provide mathematical formulations if needed. [7 pt]

B. In "Partial Image-based Puzzles" (see slides), what is the advantage of using sub-puzzles over one large puzzle, providing formulation and its rationale? [5 pt]

C. In slide 19, we describe a new puzzle mechanism that permits the pre-computation of puzzles and puzzle sharing (i.e., the same puzzle can be used by different clients). Please elaborate on at least three design rationales (discussed in the class) that permit this feature. [8 pt]

Write your solution to **HW2_Group(number).pdf**

Common Assignment Notes:

- Overall you should be submitting five files total:
  - **server.c** for puzzle generation (Q1)
  - **client.c** for puzzle solving (Q1)
  - **verify.c** for solution verification (Q1)
  - **mht.c** for offline/online phase (Q2)
  - **HW2_Group(number).pdf** for Extra Credit document
- For your convenience, we have provided script and template files with necessary functions and descriptions:
  - **UsefulFunctions.c** (Q1, Q2)
  - **VerifyYourSolutionClientPuzzle.sh** (Q1)
  - **VerifyYourSolutionMHT.sh** (Q2)
- You can refer to the OpenSSL library documentation `https://www.openssl.org/docs/manmaster/man3/` to learn how to use the functions.
- The code that throws errors or does not successfully compile/run on our side, receives zero credit (It doesn't matter if it works on your computer!).
- Follow all specific file names mentioned above, otherwise there will be 20% deduction for inconvenience.